Test Implementation (integration)

A. Hamam

Overview

- “Real” tests as soon as possible
  - Move from test abstraction to executing tests on code
- To do so
  - all the code must compile
  - Tests must not cause collateral damage
  - Process must be repeatable
  - Complete in a timely manner
- Unit tests usually passes the above, but other test phases this is a challenge
- S/W composed of many pieces of software of varying sizes (components)
- Programmers are often responsible for testing the lowest level components
- After that, testing must be carried out in collaboration with software integration

Integration testing

- Software can be integrated in many ways
- Integration testing is the testing of incompatibilities and interfaces between components (working components)
- Integration testing is carried out with incomplete systems
  - Two components in the system
  - Before the full system is complete
  - Putting the system together piece by piece and evaluating how each component fit with previous integrated component.

Integration Order

- Determine order of components
- Components depend on each other in various ways
  - Use methods of another class
  - Inheritance
  - Aggregation
- If class A uses methods defined in class B, and B is not available
  - We need test doubles
  - OR develop and test B first (no need for doubles)

CITO – Class integration test order

- A problem , still in research
- Agile for e.g. determine which features to develop next
  - Some functionality may be missing to make the system run
- Without component cycles , CITO simpler
  - Develop components that do not depend on each other first
  - Could build dependency graph – Topological Sorting
- With Cycles need to break the cycle
  - E.g.

Test Doubles

- A test double is a software component that implements partial functionality and is used in place of the “real” software component during testing
- Used to
  - Replace missing components during testing
  - Replace components with unrecoverable actions
    - E.g. A financial transaction
  - Replace components that interact with unpredictable resources
    - E.g. network connections
  - Speed
    - Slow tests might access external database, doubles uses local memory
- Writing test doubles takes effort
  - Could be incorrect too !!
- Test doubles must be integrated with little change
  - Must break dependency between component being doubled and other software components

Stocks and Mocks: Variation of Test Doubles

- Scaffolding
  - Extra software components
- Two types of scaffolding
  - Test driver: software component or test tool that takes care of the control and/or the calling of software component (e.g. JUnit framework)
  - Test doubles (stub): a skeletal or special purpose implementation of a software component used to develop or test a component that calls the stub or depends on it. It replaces a called component.
- Test stubs return a value (different from the actual component)
  - Constant values (or sometimes Null for objects)
  - Hand crafted constants, values from table lookup , random values or let the user enter return values during execution (NOT possible in Junit)
  - Test tools include automated stubbing.
- Programmers also generate their own stubs when carrying out their own unit or module testing

Stocks and Mocks: Variation of Test Doubles

- Mock is a special purpose class that includes behavior verification to check that a class under test made the correct calls to the mock
- Java reflection (specify rather than program limited behavior)
- Interaction based testing define success by how objects communicate with each other
  - As opposed to what objects do.
- NOT whether an object did the job right, but whether it was asked to do its job.
- E.g. messaging system that sends email messages to customs.

Stocks and Mocks: Variation of Test Doubles

- The model for using interaction-based testing in unit tests is:
  - Obtain necessary test doubles, possibly with a mocking tool
  - Specify the expected sequence of interactions with the test doubles
  - Carry out the action under test
  - Verify the expected interactions did occur.

Using test doubles to replace components

- How to do this without creating a configuration management nightmare?
- One option is to build separate version of the system where test doubles replace dependency components as necessary
  - Doesn't scale well if you have many test doubles
  - Test engineers must synchronize different versions (with developers)
  - If the real system sand the test system diverges it's a problem
- A safer way is to use a Seam
  - A variable that is defined in such a way that Junit test can change the program behavior without changing the s/w component that contains the variable
  - A seam needs an enabling point (a place where it is possible to set the variable to a different value
  - Control the behavior at the seam
  - Change the value of the variable from outside the s/w component (e.g. Junit test)

A simple example

- E.g.